CHAPTER 4 WORKING WITH PROGRAM MODULES

The creation of a program module is a fairly straightforward process once you have fully defined the module interfaces. However, the existence of modules would be of little value if you did not have the software tools necessary for manipulation of the modules to build your entire program. Three basic tools are required:

- A linking program that combines many modules into a single module.
- A locating program that assigns absolute memory addresses to a relocatable module.
- A library management program that permits you to add modules to a library whose contents may be accessed when linking.

These capabilities are provided by three commands LINK, LOCATE, and LIB; a standard object code format; two resident compilers, PLM80 and FORT80; and a resident assembler, ASM80.

NOTE

The location and linkage features defined in this chapter are not applicable to the object code of the 8048 microprocessor. For information about MCS-86 object module management, see *MCS-86 Software Development Utilities Operating Instructions for ISIS-II Users*.

The object code format supported by ISIS-II is a relocatable format produced by PLM80, FORT80, and ASM80. ISIS-I formats must be converted to the new format for use with ISIS-II. See the HEXOBJ and BINOBJ command descriptions in Chapter 3.

The LINK program combines files containing object modules of a program into one module in one object file, adjusting the relative addresses in the process. When the modules to be combined reference each other, you must identify these references as public symbols and external symbols thus allowing LINK to satisfy the external references of each module.

The relative addresses assigned to relocatable modules are converted to absolute addresses by the LOCATE program, which produces an absolute object file (or module that can be loaded by ISIS-II for execution, debugging, or memory mapping).

The LIB program creates and maintains libraries of object modules that can be used as building blocks to create new programs via LINK.

The many reasons for relocating a program and for writing programs in modules are discussed in the following text.

Microprocessor Memory Allocation

The microprocessor memory for any given application is generally not of uniform composition. The memory is usually tailored, with RAM (read-write memory) installed for variable data and ROM (read only memory) or PROM (programmable read only memory) installed for program code. Memory chips can be installed in such a way that some addresses have no corresponding physical memory.

This diversity in memory design (as well as programming considerations explained later) requires you to tailor your program, specifying absolute addresses for the actual ROM and RAM locations used for code and data. However, you may not know the absolute addresses of the ROM or RAM in your final application at the time you start to develop the program. At an early stage of development when you are testing your program in the system, your only concern with locating the program is that it doesn't overlap the resident routines of ISIS-II. The system memory is simply a sequence of RAM locations from 0 to 32K, 48K, or 64K. However, as your program development continues and debugging becomes a prime concern, address requirements are much more specific. Code may be in PROM starting at location 0; variable data may begin in the first location of a block of RAM.

The program that had a base address of 4000H for compatibility with ISIS-II resident software and had variable data immediately following code, may need new addresses to meet the requirements of the memory in the final application. In a system that deals only with absolute code, you would have to change the source program to specify new addresses and translate again to get object code with correct addresses. In the relocation and linkage system of ISIS-II, you simply produce a new absolute memory image from the relocatable object modules by using the LOCATE program. LOCATE assigns addresses to place code in ROM or PROM, variable data in RAM, and the stack in another area of RAM.

Program Segments

The LOCATE program allows the user to preassign areas of memory because of the way the language translators (PLM80, FORT80, and ASM80) divide a relocatable object module into segments. The segments are:

- Code
- Data
- Stack
- Memory
- Named and unnamed common segments (FORT80 only)

Each segment starts with a relative address of zero. A base address can be specified for each segment when it is being LOCATEd. The base address is the actual address of the first memory location the segment will occupy. LOCATE adds the base address to each relative address and adjusts address references accordingly.

Code Segment

The code segment is that part of the program destined for ROM because it contains machine instructions and program constants that are never modified during execution. This segment can also be placed in RAM. In assembly language it consists of statements following a CSEG directive.

Data Segment

The data segment is that part of the program that usually requires RAM. It contains variable data and storage for I/O buffers. In assembly language it consists of statements following a DSEG directive.

Stack Segment

The stack segment is for the program stack and must be in RAM. Usually only a module that contains a main routine has a reference to a stack segment. Its length is determined by the compiler for PL/M programs and by the STKLN statement in assembly language. You can also specify the length when the program is made absolute by LOCATE. References to the stack segment are made with STACK (a reserved word in assembly language) and STACKPTR (an identifier in PL/M).

Memory Segment

The memory segment is assigned to RAM memory that is not allocated to code, data, common, or stack segments. References to the memory segment are made with MEMORY (a reserved word in assembly language and an identifier in PL/M). Although the language translators create a memory segment for each relocatable object module, its length is unknown until an absolute module is produced by LOCATE, which uses the Monitor MEMCK routine and the base address of the memory segment to calculate the length of available RAM.

Common Segments

The common segments are used for FORT80 named COMMON areas and the BLANK COMMON area. Common segments usually contain variables, and therefore, are usually placed in RAM.

Absolute Information

In addition to the relocatable code, data, stack, common, and memory segments, an object module can contain information with absolute addresses already assigned. There are three ways this can happen. The ASEG statement in assembly language causes statements following it (until a CSEG or DSEG is encountered) to have absolute addresses. Absolute modules produced by LOCATE can be linked with relocatable modules. PL/M variables declared with the AT attribute produce absolute references.

It is possible to write a self-contained program in assembly language using the ASEG statement. This results in an absolute module that can be executed directly after assembling because the assembler output is a memory image. This approach is possible only in cases where relocation is not needed.

Modular Program Development

Most programs are too lengthy or too complex to code as a straightline program. You can make the job simpler by designing first a main routine that calls separate functions in subprograms. The exchange of parameters with these subprograms can be determined during the design of the main routine. However, the actual coding of the subprograms can be left until later. The final complete program is built from the main routine and the subprograms by the LINK program. The advantages of this approach to program design are summarized in the following paragraphs.

Faster Program Development

Your program can be developed faster if the modular approach is used because small modules are easier to understand and simpler to program. Breaking a program up into functional modules makes it possible to assign the pieces to a team of developers. With the specific tasks well defined, you can concentrate on programming and testing a specific module. You can supply the input required by the module and verify it by examining the output. As the modules are debugged, they can be put in a library file using the LIB program. When all modules are ready, the LINK program is used to combine the modules into one complete module, which can then be assigned absolute memory addresses by LOCATE.

Use of Different Source Languages

The modules that make up the final program need not be translated from the same language. PL/M, FORTRAN, or assembly language can be used, whichever suits the task best. Relocatable modules produced by the compiler, the assembler, or both can be input to LINK to build a program.

Shared Subprograms

When modules of one program have been tested they can be used by other programs. This means that part of the job of future programming is already done. Because the module has relocatable addresses, it can be combined with different programs, having different address requirements each time.

Easier Debugging and Program Modification

It is easier to narrow down the location of a bug when a program is divided into modules. When you have identified the module containing the error, you can concentrate on debugging that module. When a program must be modified, it may mean that only one or two modules must be changed or added. When the new or changed modules have been translated and debugged, a new absolute memory image can be created simply by using LINK and LOCATE rather than retranslating the entire source program.

Mechanics Of Relocation And Linkage

LINK is able to combine modules, adjusting relative addresses, and LOCATE is able to convert relative addresses to absolute addresses because of information put in the object modules by the translators, ASM80, PLM80, and FORT80. The following types of information in the object module are used by LINK and LOCATE:

- Relative addresses of instructions and data.
- A list of address fields in instructions or data that refer to a location in the same segment (intra-segment references).
- A list of address fields in instructions or data that refer to locations in other segments in the same module (inter-segment references).
- A list of address fields in instructions or data that refer to locations not contained in the same module (external references).
- A list of symbols in the module that are declared public or external in the source code.

You should understand external references and declaring symbols public and external to successfully use LINK and LOCATE. However, an understanding of the other topics in this section is not as important. It is provided here to give a complete picture of the mechanics of relocation and linkage.

Relative Addressing

The relative addresses of instructions and data in the code and data segments are assigned by ASM80 and PLM80 when a source module is translated. The addresses are determined by a location counter starting with zero at the beginning and incremented by the number of bytes in each instruction. The addresses are "relative" to the beginning of the segment.

LINK combines input modules to form one output module by combining all code segments into one code segment and all data segments into one data segment. The relative addresses of the first segment remain unchanged, but LINK changes the relative addresses of the segments that follow to reflect their relationship to the beginning of the new segment. In general this means adding the length of the first segment to the relative addresses of the second, adding the combined length of the first two segments to the relative addresses of the third segment, and so on.

LOCATE produces an absolute module from a relocatable one by adding the base address of each segment to each relative address in that segment to get the absolute address. The base address of each segment can be specified in the LOCATE command or left for LOCATE to assign.

Intrasegment References

In addition to relocating load addresses, relative addresses contained in instructions or data items must be adjusted. If the address refers to a location in the same segment, it is called an intrasegment reference. A jump instruction that refers back several instructions to the beginning of a loop is this kind of reference. The value put in the address field by the translator is simply the relative or absolute address of the location referred to. If it is a relative address, it is adjusted by LINK when segments are combined and finalized by LOCATE by adding the base address of the segment.

Intersegment References

When an address in an instruction refers to a location in another segment of the same module, it is called an intersegment reference. An example is an instruction in the code segment that refers to a variable in the data segment.

When LINK combines segments to produce a new object module, intersegment references are changed to reflect the new relative addresses of the locations in the other segments.

LOCATE converts the relative address of an intersegment reference to an absolute address by adding it to the base address of the segment to which reference is made.

External References and Public Symbols

When an address field in an instruction refers to a location not contained in the same module, it is called an external reference. This reference differs from those above because the translator knows nothing about the relative location of this symbol. Therefore, you must declare these symbols external. They then become known as external symbols, which means they are defined in other modules. The instructions that refer to them are external references.

The module containing an external reference is said to be an "unsatisfied" module or is said to contain an "unsatisfied" external reference. LINK combines this module with the module that contains the proper "connector". This connector is a public symbol that matches the external symbol. A public symbol is a symbol declared to be public in the source module and put in the object code with its address by the translator.

When LINK "connects" two modules by matching an external symbol to a public symbol, the value of the public symbol (its relative or absolute address) goes in the address field of the instruction that refers to it. Then LINK removes the external references. It is replaced with an intersegment or intrasegment reference if the public symbol has a relative address. If the public symbol has an absolute address, nothing replaces the external reference because no further address adjustment is required.

If the module that LINK produces contains unsatisfied external names, LINK issues a warning message about each one. This module must be linked again with modules containing the matching public symbols in order to produce a satisfied module. The unsatisfied external name messages from LINK do not indicate that an error exists. In intermediate steps of development before all modules of the program are complete, you can expect LINK to produce these messages. Also, if you have declared a name external but never make a reference to it in your program, LINK produces an unsatisfied external name message even though no unsatisfied external reference exlists.

This points up the fact that you should have some way of identifying the state of the object code in a file. Saving the memory maps from a LINK and LOCATE is one way of keeping track; using an extension in the filename that reflects the type of contents is another way.

When a module contains no external references, it is said to be "satisfied". The public symbols are not removed from the object module because they may be needed later if a new module is added that has an external reference to one of the public symbols.

If LOCATE finds an external reference in an object module it is processing, it issues a warning message but continues to produce the absolute module. The absolute module can be executed, perhaps in debug mode with a breakpoint specified to stop processing before the instruction containing the unsatisfied external reference is reached. If that instruction is executed, results are unpredictable because the address in the instruction is undefined.

Use of Libraries

Libraries make your job of building programs from object modules via the LINK program even easier. The library manager program LIB creates and maintains files containing object modules. LIB creates a directory of the modules in each library file to keep track of the modules it contains.

The LINK program treats library files in a special way. If you specify a library file as input to LINK after specifying the modules to be included, LINK searches the library for modules that contain public symbols to match the unresolved external references in the preceding modules. If a module from the library is included but it also has unresolved external references, LINK searches the library again trying to find the module with the public symbols to satisfy the new external references. This process is repeated until a search has been made to satisfy all external references.

The library manager program can give you a list of the modules in a library file, including the public symbols in each module. You may want to keep all the object modules for one program in a library, or you may want to keep modules relating to a specific function in one library file. For example, a system library supplied with ISIS-II is called SYSTEM.LIB and contains the object modules for linking programs to ISIS-II system routines.

Link Command

The ISIS-II program LINK allows you to combine object modules from several input files into one object module in one output file. In the process of combining modules LINK adjusts all addresses so they are relative to the beginning of the segments in the output module. LINK also searches libraries for modules that resolve external references in the modules being combined and includes them in the output module. If any unresolved external references remain in the output module, LINK puts a message in the map that describes the structure of the new module.

The output module must be processed by LOCATE before it can be executed. The LOCATE program assigns absolute memory locations to the object module. The output module can also be used as input to LINK to be combined with other modules into a new and expanded output module.

The LINK program is called into operation by the LINK command. The syntax of the LINK command is:

LINK <inputlist> TO <outputfile> [<controls>]

where

<inputlist> can be either or both of the following two items:

<filename> [(<modname1>,<modname2>,...,<modname*n*>)]

PUBLICS(<filename>,<filename2>,...,<filename*n*>)

In the first item <filename> specifies a file containing object modules or a file containing a library of object modules. If <filename> is not a library file, it is included in the output module. If <filename> is a library file and <modnames> are specified, then only the specified modules are linked into the output module. If <modnames> are omitted and <filename> is a library, only those library modules that satisfy external references in modules already named in the <inputlist> or already included from the library are linked into the output module. In other words, when <modnames> is omitted, only those library modules that satisfy an existing unresolved reference are included.

The second item PUBLICS specifies modules whose absolute public declarations only are to be included in the output module. This allows for linking modules without combining them in the output file so they can be loaded separately. See the section on overlays in this chapter for a description of this capability.

<outputfile> specifies the file to contain the object module resulting from linking the input modules. This file must not also be specified in the input list.

<controls> are one or more keywords that control the operation of LINK. The controls are:

MAP	This control requests that a link map be produced. The link map is sent to the console output device (:CO:) or to the file specified in the PRINT control. The content of the link map is described later in this chapter.

NAME(<modname>) This control specifies the name to be assigned to the output module. The name can be from 1 through 31 characters, each of which must be a letter (A through Z), a digit (0 through 9), a question mark (?), or a commercial at sign (@). However, the first character of the name cannot be be a digit. If NAME(modname) is not specified, the name part of the output file specification is used as the module name.

PRINT(<filename>) This control specifies the file to contain the link map. If omitted, the link map goes to the console output device (:CO:).

If a LINK command is longer than one line on your console (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and the associated parameter list, for example:

```
PRINT&
(:F1:PRTFIL)
```

is a valid use of the continuation character. LINK prompts for the continued line with a double asterisk (**). If necessary, subsequent lines can be continued also.

WARNING

LINK uses a temporary file named LINK.TMP on the disk to which the output is directed. If you have a file by this name on the output disk, it will be destroyed.

"Appendix C: Error Messages" lists the LINK error messages.

Link Map

The link map produced by LINK includes the following information:

- The LINK sign-on message.
- The command used to call LINK (unless map is output to :CO:).
- The length of the relocatable segments in the output module.
- The addresses of absolute information in the output module.
- The names of the input modules.
- Unresolved external names.
- Non-fatal error messages.

The following example shows a link map for the output module contained in the file SHIP.WRK, which was produced from two modules explicitly listed in the input list, a module included from searching a library to resolve external references, and the public symbols of a fourth module.

The REL column in the LINK map specifies the segment relocation type. The following abbreviations are used in the REL column:

B byte relocatable

P page relocatable

I in-page relocatable

A absolute or non-relocatable

```
ISIS-II OBJECT LINKER Vx.y INVOKED BY:
—LINK FILNAM.EXT,PROG.LIB(TRIG),PROG.LIB,&
**PUBLICS(TANSTA.AFL) TO SHIP.WRK NAME(CELESTIGATION) MAP
```

(Non-fatal error messages appear here.)

```
LINK MAP OF MODULE CELESTIGATION
WRITTEN TO FILE :F0:SHIP.WRK
MODULE IS NOT A MAIN MODULE

SEGMENT INFORMATION:
  START      STOP       LENGTH       REL    NAME

                         2345H        P     CODE
  10BCH      10FFH         44H              CODE        *GAP*
  22FEH      22FFH          2H              CODE        *GAP*
                          107H        B     DATA
                           75H        B     /FRED/
                         10F2H        B     //
  0000H      0002H          3H        A     ABSOLUTE
  0040H      0711H        6D2H        A     ABSOLUTE
  0700H      07FFH        100H        A     ABSOLUTE    *OVERLAP*
  FD00H      FD00H          1H        A     ABSOLUTE

INPUT MODULES INCLUDED:
:F0:FILNAM.EXT (NAVPACK)
:F0:PROG .LIB (TRIG)
:F0:PROG .LIB (EXP)
:F0:TANSTA.AFL (MCBRIDE) (PUBLICS)

UNRESOLVED EXTERNAL NAMES:
   COSIGN—REFERENCED IN :F0:FILNAM.EXT(NAVPACK)
```

(Other errors appear here.)

Order of Modules in the Output File

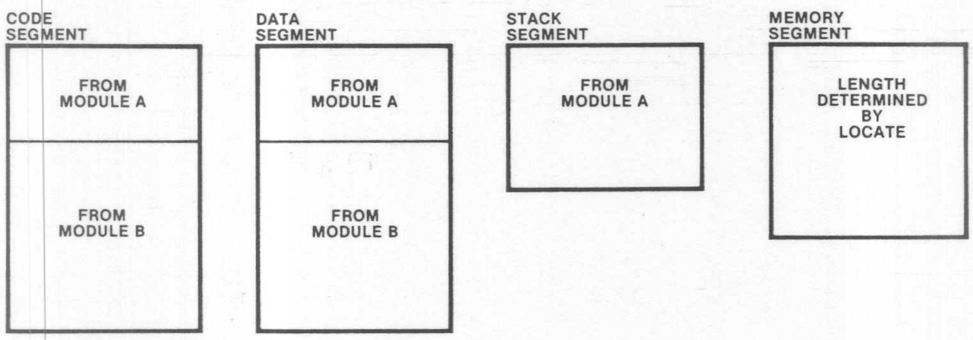
LINK combines modules from the input list by combining all the code segments from the input modules into one code segment, all the data segments into one data segment, and all the stack segments into one stack segment. The length of the memory segment is computed by LOCATE. Any absolute information in the input modules is transferred to the output module with absolute addresses unchanged. If absolute information is in conflict for the same location, a message to that effect is put in the link map.

The order of combining follows the order in which modules are specified in the input list. The first module specified is the first in order in combining. The segments of the second module follow the segments of the first module at the first available location.

Example : The following LINK command and explanation shows how modules are combined.

LINK A,B TO C

Module A contains code, data, and stack segments. Module B contains code and data segment. They are all byte relocatable. The resulting module C has the following structure



In the process of linking the input modules, external references are satisfied. For instance, if module A has a branch to a point in module B, it is no longer external when A and B are combined into C. If A or B has an external reference to a module in a library file, the library must be specified in the input list *after* the module that refers to it. Suppose the library file RTNS.LIB contained a module that satisfied an external reference in A. Then the input list could be specified A,RTNS.LIB,B or A,B,RTNS.LIB. "Appendix C: Error Messages" lists the LINK error messages.

Locate Command

The LOCATE program takes an input file containing a relocatable object module and produces an output file containing the object module with the relative addresses fixed to absolute locations. The LOCATE program is activated by the LOCATE command. The syntax of the LOCATE command is:

LOCATE <inputfile> [TO <outputfile>] [<controls>]

where:

<inputfile> is the name of the file containing the relocatable object code.

<outputfile> is the name of the file that is to contain the absolute object module. If TO <outputfile> is omitted, it uses the filename portion of <inputfile>. If a file already exists with the same name as specified by <outputfile> (or the defaulted name), it is overwritten with the new data. If <outputfile> is not specified, <inputfile> must consist of a filename and extension because the default name for the output file is the filename (without extension) from <inputfile>.

<controls> specifies one or more keywords that control the operation of LOCATE. The controls and their default values are explained in the following descriptions. The LOCATE controls are:

MAP
COLUMNS(number)
PRINT(file)
SYMBOLS
LINES
PUBLICS
PURGE
ORDER(segment sequence)
CODE(address)
DATA(address)
STACK(address)
MEMORY(address)
/common name/(address)
//(address)
NAME(name)
RESTART0
START(address)
STACKSIZE(value)

If a LOCATE command is longer than one line on your console, (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and the associated parameter list, for example:

```
STACKSIZE &
(40)
```

WARNING

LOCATE uses a temporary file named LOCATE.TMP on the disk to which the output is directed. If you have a file by this name on the output disk it will be destroyed.

"Appendix C: Error Messages" lists the LOCATE error messages.

LOCATE Control Descriptions

MAP

The MAP control specifies that a memory map be printed on the list device. Normally, the printing of the map is suppressed. The map lists the start address for the module, the start and stop addresses for each segment along with their length, and relocation type.

The map lists the start and stop addresses of the segments and length of each segment. The REL column specifies the relocation type. The following abbreviations are used in the REL column:

B	byte relocatable
P	page relocatable
I	in-page relocatable
A	absolute or non-relocatable

When a segment overlaps another segment, a warning, (OVERLAP) is printed in the map. This does not stop the locate function. This may be intentional, as in the sample map. The three byte absolute segment is designated to fit into the data segment.

The following is an example of a memory map:

```
ISIS-II OBJECT LOCATER Vx.y INVOKED BY:
-LOCATE TRIG.REL MAP PRINT(:CO:) NAME(TRIG)

MEMORY MAP OF MODULE TRIG
READ FROM FILE :F0:TRIG.REL
WRITTEN TO FILE :F0:TRIG
MODULE START ADDRESS 3000H

    START     STOP      LENGTH    REL   NAME

    0008H     000AH         3H     A    ABSOLUTE
    3000H     343FH       440H     B    CODE
    3440H     472EH      12EFH     B    DATA
    3630H     3632H         3H     A    ABSOLUTE (MEMORY OVERLAP)
    472FH     475FH        31H     B    STACK
    4760H     F6BFH      AF60H     B    MEMORY
```

WARNING

The length of the MEMORY segment is always computed to be the amount of available memory on the host Intellec development system. If the module is executed on the Intellec system, the MEMORY segment length is correct as specified. If the module is executed on a different system the actual amount of memory depends on the configuration of that system. LOCATE has no knowledge of the configuration of the target system.

COLUMNS(number)

The COLUMNS control specifies whether the symbol table in the list file is to be printed in 1, 2, or 3 columns. The default is 1. This control is ignored unless SYMBOLS, LINES, or PUBLICS is specified.

PRINT(file)

The PRINT control specifies a list file for the LOCATE program output. If the PRINT control is not specified, the output goes to the console output device (:CO:).

SYMBOLS

The SYMBOLS control specifies that a list of local symbols (within a module) and input module names is to be included in the symbol table in the list file. Normally, the printing of the symbol table is suppressed. Local symbols are listed in the table with a type of SYM and module names have the type MOD.

The following is a sample symbol table:

```
SYMBOL TABLE OF MODULE TRIG
READ FROM FILE TRIG.REL
WRITTEN TO FILE TRIG

 VALUE      TYPE      SYMBOL

            MOD       TRIG
 3011H      PUB       SIN
 3015H      SYM       SIO
 3027H      SYM       SI1
 3040H      SYM       TRIGEX
 3063H      PUB       COS
 3102H      PUB       TAN
 3230H      SYM       TAN0
 3340H      LIN       272
```

LINES

The LINES control specifies that a list of the line numbers and input module names from the program is to be included in the symbol table in the list file. Line numbers have the type LIN and module names have the type MOD. The VALUE field is blank for module names in the symbol table. Normally, the printing of the symbol table is suppressed.

PUBLICS

The PUBLICS control specifies that a list of symbols declared public is to be included in the symbol table. Public symbols have the type PUB. Normally, the printing of the symbol table is suppressed.

PURGE

The PURGE control specifies that line numbers, local symbols, module names, and public symbols be removed from the output module. Public symbols could be used, if present, to link the absolute module with other modules. Line numbers, local symbols, and module names could be used for debugging. The PURGE control condenses the size of a module that is completly debugged. This results in saving disk space and decreasing load time.

ORDER(segment sequence)

The ORDER control defines the order in which the various segments are assigned memory locations. The list of segments in the segment sequence parameter must be separated with spaces.

If ORDER is not specified, the segments are located in the following order:

```
CODE
STACK
/commons/ (named and unnamed commons in an arbitrary order)
DATA
MEMORY
```

This default order can be changed with the ORDER control. LOCATE determines the addresses at which the segments reside but it is done in the order specified by the control.

A partial list can be specified in the ORDER control. When a partial list is specified, the segments specified are located first in the order specified. The remaining (unspecified) segments follow in the default sequence. For example, the control:

```
ORDER(DATA)
```

results in the sequence: DATA, CODE, STACK, /commons/, MEMORY. (Assuming all segment types exist in the module.)

The first segment is located 680H bytes above the top of the ISIS-II code (3680H). This allows room for 13 input and output buffers (six open files and :CO: and :CI:). You can also change the order with the CODE, STACK, /name/, //, DATA, and MEMORY controls by specifying a specific address at which each will reside. The section "How LOCATE Locates Segments," following the control descriptions describes how the default order, ORDER control, and specific address controls interact to locate segments.

```
CODE(address)
DATA(address)
STACK(address)
/common name/(address)
//(address)
MEMORY(address)
```

The segment locations can be specified explicitly with the segment controls. The controls are specified with the address. The address can be in decimal, hexadecimal, octal, or binary. The address must begin with a digit and may be followed by a letter specifying the base of number:

Decimal - D or omitted
Hexadecimal - H
Octal - O or Q
Binary - B

The specified addresses of some segments may be changed by LOCATE. If because of the addressing within a segment, it must reside at or between 256 byte multiples of memory, the Locate program will make the adjustment.

The section "How LOCATE Locates Segments," following the control descriptions describes how the default order, ORDER control, and specific address controls interact to locate segments.

NAME(name)

The NAME control specifies a name for the output module. The name can be from 1 through 31 characters, each of which must be a letter (A through Z), a digit (0 through 9), a question mark (?), or a commercial at sign (@). However, the first character of the name cannot be a digit. If NAME is not specified, the name in the header record of the input file is used.

RESTART0

The RESTART0 control places a jump instruction at locations 0, 1, and 2 in the absolute module. The address in the jump instruction is the programs starting address (the address of the first instruction to be executed) taken from the input module or from the START control. You would use the RESTART0 control when

preparing an absolute module for execution in your prototype system, either the standalone system or the system being emulated with the in-circuit emulator. When the RESET signal is input to the CPU, the program counter is set to 0 and execution begins with the jump to the beginning of your program. An absolute module prepared with the RESTART0 control is not compatible with ISIS-II, which does not allow user code to be loaded in locations 0, 1, or 2. The RESTART0 control is ignored if the input module is not a main module.

START(address)

The START control specifies the address of the first instruction in the code segment to be executed. This address overrides the address in the input module. If START is omitted, the address is taken from the input module. The START control is ignored if the input module is not a main module.

STACKSIZE(value)

The STACKSIZE control specifies a value (in bytes) for the size of the stack segment. This value overrides any calculated stack size encountered in the input module.

When debugging a program in an Intellec microcomputer development system, 12 additional bytes of user stack are required beyond that computed by the language translator or LINK. LOCATE adds these 12 bytes if the STACKSIZE parameter is not specified.

How LOCATE Locates Segments

Module segments are normally located sequentially in memory in the order:

CODE segment
STACK segment
/commons/ segments (in an arbitrary order)
DATA segment
MEMORY segment

You can change the order with an ORDER control and with the CODE, STACK, /common/, //, DATA, and MEMORY controls. You can change the order by using the default order in conjunction with the ORDER control and the segment controls specifying an exact address.

Locating With The Default Order

When you use the default order, the CODE segment is located 680H bytes above the top of ISIS-II and the rest of the segments immediately follow in order. Gaps are generated only when required by the relocation type of the segment.

- Byte relocatable (BR) segments are located at the first available byte.
- Page relocatable (PR) segments are located at the first available byte that lies on a page boundry (a multiple of 256 (100H) bytes).
- In-page relocatable (IP) segments are located at the first available byte such that the segment is totally contained within a page.

Locating With the Default and ORDER Control

You can change the order totally with the ORDER control. When you specify all segments in the ORDER control the first segment is located 680H bytes above the top of ISIS-II and the rest of the segments immediately follow in the order specified in the ORDER control. Gaps are generated only when required by the relocation type of the segement.

If you don't specify all the segments in the ORDER control:

- First, the segments specified in the ORDER control are located in the order specified.
- Next, the segments not specified in the ORDER control are located immediately following the last segment specified, in the default order.

Gaps are generated only when required by the relocation type of the segments.

If you submit the following ORDER control with the LOCATE command:

```
ORDER(DATA STACK)
```

the segments will be located in the following order:

DATA
STACK
CODE
/commons/ (if FORTRAN)
MEMORY

Locating With the Default, ORDER Control, and Specific Addresses

You can change the order with a combination of the default order, an ORDER control, and specific segment location controls. You should not specify a segment in the ORDER control and with a segment location control. When all three forms of locating are used, segments are located according to the following:

- Segments are selected for placement in the order specified by the ORDER control and the default sequence as described in the preceding section.
- The starting address of a segment is either the address following the preceding segment (680H above ISIS-II for the first segment) or the address specified in a location control. A gap will be generated if required by the relocation type of the segment.

If the LOCATE command is submitted with the following controls:

```
ORDER(DATA STACK) CODE(6000H)
```

the segments are placed in the sequence:

DATA
STACK
CODE
/commons/
MEMORY

The DATA segment is located 680H bytes above the top of ISIS-II and STACK is located immediately following DATA. CODE is located next, but instead of being placed immediately following STACK it is placed at address 6000H as specified by the CODE control. The /commons/ (if any) immediately follow CODE and MEMORY follows the /commons/. In other words, the segments are still located in the sequence specified by the default order and the ORDER control but the locations are modified by the individual segment location controls.

If you are going to locate some segments at specific addresses and let LOCATE place the rest, you should use the ORDER control to modify the default sequence so that segments are located in an order that corresponds with the specific addresses in the controls. If you aren't careful, conflicts can occur. Be sure to specify the MAP control, to verify that segments are placed as intended. Conflicts do not stop the LOCATE function, because there are circumstances where you will want apparent conflicts such as the location of an absolute segment in an internal gap in a segment.

When you want to locate FORTRAN common segments to specific addresses, you should also locate the MEMORY segment to an address above the top of the highest common segment. LOCATE handles the common segments in an arbitrary order. You will not know ahead of time what order the common segments will be handled by the command. If the common segment that you place at low memory is the last one handled by LOCATE, the MEMORY segment will immediately follow it and will conflict with all segments above it.

LIB Command

WARNING

LIB uses a temporary file named LIB.TMP on the disk to which the output is directed. If you have a file by this name on the output disk it will be destroyed.

The ISIS-II LIB program allows you to create specially formatted files to contain libraries of object modules, to maintain these libraries by adding and deleting modules, and to obtain a listing of the modules in a library file. Libraries can be used as input to LINK, which may automatically link modules from the library that satisfy external references in the modules being linked.

The library manager program is called into operation by the LIB command. The syntax of the LIB command is:

LIB

The operation of LIB is controlled by entering commands to indicate which operation LIB is to perform. LIB prompts for commands with an asterisk (*). The commands are:

CREATE
ADD
DELETE
LIST
EXIT

Continuation Lines

If a command to LIB is longer than one line on your console (which must not be greater than 122 characters), you can continue it by entering an ampersand (&) as the last non-blank character before the carriage return. The ampersand cannot appear within a filename or control keyword. It can be placed between a keyword and a parameter, for example:

```
DELETE PVT.LIB&
(MOD1)
```

LIB prompts for the continued line with a double asterisk (**). If necessary, subsequent lines can be continued also.

CREATE - Create a Library File

The CREATE command creates an empty library file. You must use the ADD command to add modules to the library file. The syntax of the CREATE command is:

CREATE <filename>

where

<filename> specifies the name to be assigned to the new library file. If a file with that name already exists, an error message is sent to the console and LIB prompts for another command.

ADD - Add Modules to a Library File

The ADD command adds object modules to a library file. The syntax of the ADD command is:

ADD <filename>[(<modname>,...)] [,...] TO <libfile>

where

<filename> can be the name of a library file or the name of a file containing an object module. If a library file is specified, all the object modules contained in it are added to <libfile> unless <modnames> are specified.

<modnames> can be specified only if <filename> is a library file. Only the object modules specified by <modnames> are added to <libfile>.

<libfile> is the library file being modified by the addition of modules in <filename>.

DELETE - Delete Modules from a Library File

The DELETE command deletes modules from a library file. The syntax of the DELETE command is:

DELETE <libfile> (<modname>,...)

where

<modname> specifies the object module to be deleted from <libfile>.

LIST - List Library Modules and Their Public Symbols

The LIST command lists the module directory of the library file. The syntax of the LIST command is:

LIST <libfile>[(<modname>,...)][,...] [TO <listfile>] [PUBLICS]

where

<libfile> is the name of the library file whose entire module directory is to be listed unless <modname> is also specified. In that case, only information about the specified modules is listed.

<listfile> is the name of the file to contain the library listing. If omitted, the directory is listed on the current console output device (:CO:).

PUBLICS specifies that public names in each module are to be listed. If omitted, only the module names are listed.

The format of the listing when public names are requested is:

```
*LIST TEST.LIB PUBLICS

TEST.LIB
 |OPEN
 |    NOREX
 |    ABEX
 |REDUCE
 |    HEX
 |    OCT
 |    DATUM
 |CLOCK
 |  | TIME
 |  | LAPSE
 |  | CYC
 |  |  |_____ public names
 |  |________ module names
 |___________ library name
```

EXIT - Return to ISIS-II

The EXIT command returns control to ISIS-II. When finished with LIB, enter the EXIT command, followed by a carriage return. This terminates the LIB program and returns control to ISIS-II, which prompts for a command with a hyphen (-).

Example : The following example shows the creation of a library file and the entry in the library of two modules. The directory of the library is listed before exiting to ISIS-II.

```
-LIB
ISIS-II LIBRARIAN Vx.y
*CREATE FOO.LIB
*ADD SIN.OBJ,COS.OBJ TO FOO.LIB
*LIST FOO.LIB
  SINE
  COSINE
*EXIT
-
```

"Appendix C: Error Messages" lists the LIB error messages.

Program Overlays And Linked Loading

When a program is larger than the available memory space, it is necessary to link modules without combining them into one module. Thus during execution when part of the program is no longer needed, another part can be loaded in the same area of memory, overlaying the part not needed. Under ISIS-II, programs or parts of programs to be loaded separately must be in separate files. The first load can be done by entering the name of the file as a command. The subsequent loads are done from a program with the LOAD system call or an I/O routine.

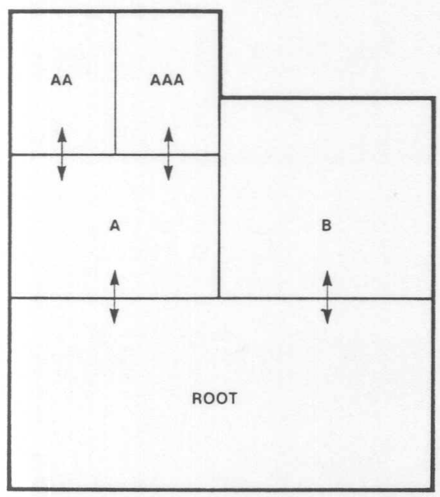
In the typical use of LINK and LOCATE, modules with external references are combined with modules that have matching public symbols to produce a module with no unsatisfied external references. In linking without combining, the external references must still be satisfied; that is, they must know the addresses of their matching public symbols. Using the keyword PUBLICS before a list of modules tells LINK that the modules are not to be combined in the output module but used only to supply the addresses of their public symbols.

In this way the external references of modules listed earlier in the input list are satisfied. For example,

```
LINK A,PUBLICS(B,C) TO A.SAT
```

results in a module A.SAT whose external references to symbols contained in B and C are satisfied. However, modules B and C must be absolute modules because LINK must know the absolute addresses of the public symbols. Therefore the typical use of LINK before LOCATE is reversed. You must LOCATE modules B and C first, creating modules with absolute addresses but perhaps with unsatisfied external references. The module created in this way can be considered a temporary module used only to supply addresses of public symbols needed by other modules.

Consider the following example. A root segment calls segment A and later calls segment AA, then segment AAA overlays AA. A diagram of this overlay structure follows where the vertical lines indicate a division in time (segments A and B are not in memory at the same time) and the horizontal lines indicate a division in memory space.



All the pieces of this overlay structure must be in separate files because they are loaded separately. The root calls A and B; A calls AA and AAA. If you locate the root first, you can use the memory map produced by LOCATE to determine the base address of A and B. Then locate A and B. Use the memory map produced by LOCATE for A to determine the base address of AA, AAA. The modules produced by LOCATE have absolute memory addresses assigned but external references are unsatisfied. These modules cannot be loaded into memory for execution but can be used with the PUBLICS keyword in the linking process.

Suppose the modules produced by LOCATE were given the extension of TMP. Suppose the root has external references to public symbols in A and B; A has external references to public symbols in the root, AA, and AAA; AA and AAA have external references to public symbols in A; B has external references to public symbols in the root. Then the following LINK commands would produce the satisfied modules ready to execute.

```
LINK ROOT.TMP,PUBLICS(A.TMP,B.TMP) TO ROOT
LINK A.TMP,PUBLICS(ROOT.TMP,AA.TMP,AAA.TMP) TO A
LINK AA.TMP,PUBLICS(A.TMP) TO AA
LINK AAA.TMP,PUBLICS(A.TMP) TO AAA
LINK B.TMP,PUBLICS(ROOT.TMP) TO B
```

The modules ROOT, A, AA, AAA, and B are absolute because of the previous LOCATE operation and fully satisfied because of the above LINK operation. They are connected but not combined.

To verify that all external references are satisfied in the modules produced by LINK, you can LOCATE them again. This time LOCATE should produce no message about unsatisfied external differences.

CAUTION

When you link without combining to produce overlays, you must provide overlay management in the design of your system. That is, before your program makes a reference to an overlay segment, it must make sure that segment is in memory. If not, the segment must be read into memory. When a segment in memory contains new data that must be saved, it must be written out before it is overlaid with another segment. The ability to link without combining provides the hooks for an overlay scheme. The runtime management of overlays must be designed into your software.

Memory Pages and the H and L Registers

Relocation types are provided for programs that reference memory by manipulating the H and L registers independently. (See the *8080/8085 Assembly Language Programming Manual* 9800301, for a description of the HIGH and LOW operators.) You can store data on a page boundary and address elements in it by changing only the L register. If the data does not cross a page boundary, you do not have to change the H register at all.

However caution must be used if the HIGH operator is used on an arbitrary address in relocatable code, you may get an incorrect address because LOCATE assumes the unused portion of the address to be zero. If the unused portion of the address is not

zero and the addition of the low order portion of the segment base address causes a carry into the high order portion, that carry will not be detected when the HIGH operator is used. For example, if HIGH is used on the relocatable address 1234H:

HIGH(1234H) = 12H

and LOCATE adds a segment base address of 10F0H:

WITH HIGH OPERATOR	WITHOUT HIGH OPERATOR
1200H	1234H
+10F0H	+10F0H
22F0H	2324H
THE HIGH PART OF WHICH IS 22H	THE HIGH PART OF WHICH IS 23H

Because LOCATE has no knowledge of the low order portion of the address there is a chance that located HIGH address will be off by one. The located address will be correct if there is no carry from the low order portion.

You can avoid this situation by only using the HIGH operator on addresses in segments that you have defined as *page relocatable* .

This circumstance does not exist with the LOW operator. Addresses on which LOW has been used will always be correct.

Savings in memory space and execution time can result from this method, but access to some areas of memory may be lost because of the way LINK and LOCATE act to preserve relocation types.

When LINK combines segments having different relocation types, it follows these rules:

- Byte relocatable segments follow the preceding segment at the next byte. The output segment is byte relocatable only if all input segments are byte relocatable. Otherwise, it is page relocatable.
- Page relocatable segments follow the preceding segment at the first available page boundary. The bytes from the end of the preceding segment to the page boundary, if any, are unused. The output segment is page relocatable.
- Inpage relocatable segments are located at the first location following the preceding segment if they fit within the page. Otherwise, they are located at the next page boundary and the bytes at the end of the previous page are unused. The output segment is inpage relocatable only if it is not more than 256 bytes and all input segments are inpage relocatable. Otherwise, the output segment is page relocatable.

The bytes that are unused in the process of preserving the relocation type of segments are flagged in the LINK memory map by the word 'gap.' These gaps are unused portions of the program; in a sense they are ''lost.

If LOCATE assigns the base addresses of segments, it does so in a way that preserves the relocation type of the segment. It also issues a message if an inpage relocatable segment is changed to page relocatable. If you specify a base address for a segment with a segment location control that violates the relocation type of the segment, LOCATE places the segment on the next higher page boundary and issues a message.